Increasing Web Server Throughput with Network Interface Data Caching

Hyong-youb Kim, Vijay S. Pai, and Scott Rixner
Computer Systems Laboratory
Rice University
Houston, TX 77005
{hykim, vijaypai, rixner}@rice.edu

Abstract

This paper introduces network interface data caching, a new technique to reduce local interconnect traffic on networking servers by caching frequently-requested content on a programmable network interface. The operating system on the host CPU determines which data to store in the cache and for which packets it should use data from the cache. To facilitate data reuse across multiple packets and connections, the cache only stores application-level response content (such as HTTP data), with application-level and networking headers generated by the host CPU. Network interface data caching can reduce PCI traffic by up to 57% on a prototype implementation of a uniprocessor web server. This traffic reduction results in up to 31% performance improvement, leading to a peak server throughput of 1571 Mb/s.

1. Introduction

Networking server performance has improved substantially in recent years, due mostly to rapid developments in application and operating system software and, to a lesser extent, improved processing power in the network interface. These developments have reduced the CPU load of network servers, the amount of main memory used for networking, and the bandwidth requirements of data transfers between the CPU and its main memory. However, the local interconnect within a server, such as the peripheral component interconnect (PCI) bus, remains a performance bottleneck in server applications since all data being sent over the network interface must be transferred across this interconnect.

This work is supported in part by a donation from Advanced Micro Devices and by the National Science Foundation under Grant No. CCR-0209174.

ASPLOS X 10/02 San Jose, CA, USA

This paper proposes and evaluates network interface data caching, a new technique to alleviate the local interconnect bottleneck by caching data directly on a programmable network interface. A software-managed data cache in the network interface reduces internal server traffic by minimizing repeated transfers of frequently-served content across the local interconnect. The cache resides in a modest amount of on-board DRAM of a network interface card with a programmable processor that can store and access data within the cache. The operating system on the host CPU determines which data to store in the network interface data cache and for which packets it should use data from the cache. Cache contents may be appended to packet-level and application-level headers generated by the host processor and then sent over the network.

A prototype implementation of a server with network interface data caching achieves benefits by both reducing PCI traffic and increasing server throughput. This prototype uses a uniprocessor PC-based server with two Gigabit Ethernet links running the *thttpd* web server [15]. The prototype server runs the FreeBSD 4.6 operating system with roughly 1000 lines of new and modified code to support the network interface data cache. The prototype server is accessed by a testbed of synthetic clients replaying existing web logs. Adding a cache with 16 MB of memory on each network interface shows 36–57% reductions in server PCI bus traffic for four out of five workloads, resulting in throughput improvements of 7–31% for three workloads and peak throughput of 1571 Mb/s. Although the prototyped system uses a PCI bus, network interface data caching does not depend on the specifics of the local interconnect.

Network interface data caching exploits techniques originally developed for zero-copy I/O schemes, which avoid duplicating data between the kernel and user space of main memory [7, 11, 13]. Caching data at the network interface essentially extends the benefits of zero-copy I/O to the final crossing in the server between the memory and the network interface card (NIC). This technique alleviates the local interconnect bottleneck, allowing application-level performance to scale with more aggressive CPUs and network

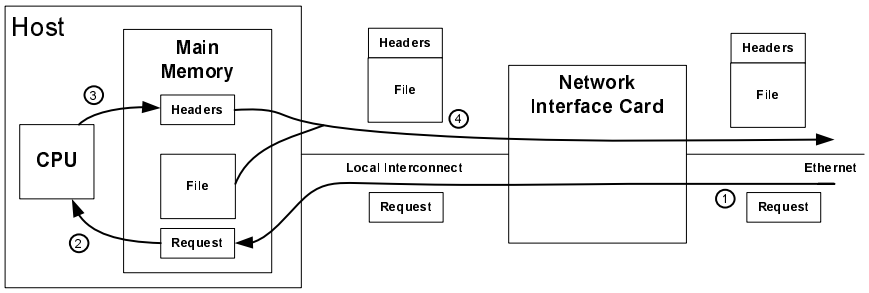

Figure 1: Steps in processing an HTTP request to a file that currently resides in the OS file cache. Step 1: the HTTP request is transferred from the Ethernet through the network interface to the main memory. Step 2: the kernel reads the request, processes it, and passes it on the web server. Step 3: the web server generates HTTP headers and tells the kernel which file to send. The kernel generates TCP, IP, and Ethernet headers, and finds the file in its file cache (if the file is not in the file cache, the file system initiates a disk read). Step 4: the headers and file are transferred to the Ethernet through the network interface.

links without the expense and redesign effort of a new interconnect.

The remainder of this paper proceeds as follows. Section 2 describes the flow of request and response data in a web server. Section 3 explains the concept of network interface data caching and its potential benefits, and Section 4 details the design and use of the cache. Section 5 describes the experimental methodology, including the prototype implementation and test platform. Section 6 discusses the experimental results. Section 7 discusses related work, and Section 8 concludes the paper.

2. Anatomy of a Web Request

To illustrate the performance issues in web servers, this section considers the flow of a request and response through a system that supports zero-copy I/O and includes a network interface that supports checksum offloading. Operating systems that support zero-copy I/O use special APIs or memory management schemes to avoid copying data between the kernel and user space of main memory or between different subsystems of the kernel (such as the file cache and network buffers) [7, 11, 13]. Network interfaces that support checksum offloading reduce load on the host CPU by directly calculating IP header checksums and TCP packet checksums, allowing the operating system to transfer packets to and from the NIC without computing any checksums [9].

Figure 1 shows the steps taken by a typical web server to process an HTTP request for a static page and produce a response, assuming the requested file resides in the file cache. In the first step, the HTTP request packets arrive on the Ethernet link connected to this server's network interface. The network interface card initiates a *direct memory access* (DMA) across the local interconnect, through which the device takes control of the interconnect and writes data into the system's main memory. In step 2, the CPU reads the request packet data from memory so that the TCP/IP stack of the kernel and the application-level HTTP server can take appropriate actions based on the request. In step 3, the application responds by creating a set of HTTP headers corresponding to the requested file and writing them into memory. The application then initiates a TCP/IP transmission of the file through a system call. On a file cache hit, the kernel passes a reference to the data in the file cache buffers to the TCP/IP stack. (On a miss, the file system would first read the file from disk.) The TCP/IP stack then writes the protocol headers to memory and has the device driver alert the NIC of a new transmission. In step 4, the NIC initiates DMA transfers of the TCP/IP headers, HTTP headers, and HTTP content from the system memory to the network interface buffers. Finally, the network interface calculates checksums for each packet and sends the data out onto the network.

Since a web request requires service from the processor, the local interconnect, the network interface, the Ethernet, and possibly the disk, each of these system components can limit the performance of the web server. Many studies have focused on improving processor efficiency by reducing the processing requirements of user requests. For example, scalable event notification and dispatch mechanisms, persistent connections, asymmetric event-driven systems, and zero-copy I/O have all dramatically improved web server throughput by reducing the application-level and kernel-level processing required by the server [3, 8, 11, 12, 13]. Similarly, many caching strategies and prefetching techniques have been developed to reduce the impact of disk latency and bandwidth limitations. Network bandwidth has improved steadily, and high-speed networks such as Gigabit Ethernet are now common in server environments. High-performance network interface cards that connect a 64-bit/33Mhz PCI bus to a Gigabit Ethernet are readily available, inexpensive, and typically include support for checksum offloading. Moreover, simply adding additional network interfaces will increase the available link bandwidth of a web server.

Although the local interconnect can also be a bottleneck, few studies have addressed its efficiency. A standard 64-bit/33Mhz PCI bus provides a peak bandwidth of 2 Gb/s. Although a single full-duplex Gigabit Ethernet link provides a maximum bandwidth of 2 Gb/s, a web server typically only

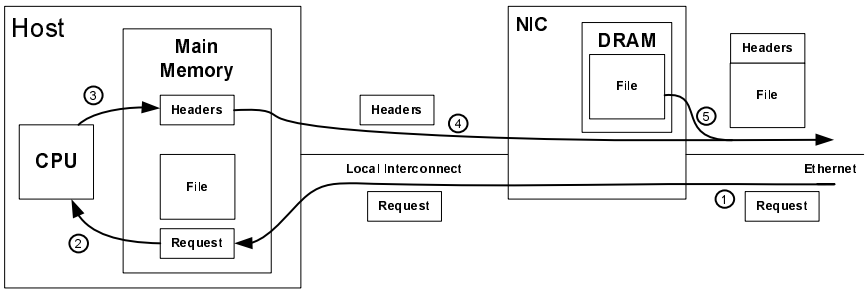

Figure 2: Steps in processing an HTTP request to a file that currently resides in the network interface data cache. Steps 1–3 are unchanged from the original system, shown in Figure 1. In step 4, after the headers are generated they are sent to the network interface without the file. The network interface then retrieves the file from its cache and transfers the headers and file out onto the Ethernet in step 5.

utilizes the half-duplex bandwidth of 1 Gb/s since the volume of outbound traffic (HTTP responses) dominates that of incoming traffic (HTTP requests). As a result, a web server as described in Figure 1 can theoretically use a 64-bit/33Mhz PCI bus to support two Gigabit Ethernet NICs at full transmit bandwidth. In reality, however, the PCI bus cannot deliver this theoretical maximum throughput because it is shared among all peripherals in the system and because it has additional control overhead beyond the headers and data that eventually reach the peripherals. Consequently, the PCI bus can become a performance limiter as the server begins to utilize a significant fraction of the Ethernet bandwidth.

3. A Network Interface Data Cache

The addition of a data cache within the network interface can improve server throughput by reducing traffic over the local interconnect. By storing frequently requested files in this cache, the server will not need to send those files across the interconnect for each request. Rather, the server can simply generate the appropriate protocol and application headers, and the network interface card can combine those headers and the file data to be sent out over the network. Storing copies of files in a cache on the network interface in this manner accelerates step 4 of Figure 1 by eliminating the final transfer of file data from the system memory to the network interface storage, reducing the bandwidth demands on both the local interconnect and main memory.

Figure 2 shows the stages in processing a web request in a system with a network interface data cache. As in Figure 1, requests arrive on the Ethernet, are transferred to main memory, and are read by the CPU to generate the appropriate headers (steps 1–3). In step 4, however, if the kernel determines that the file being sent is currently cached within the network interface, then only the headers and the location of the file in the NIC's memory are transferred to the network interface via DMA. In step 5, the network interface then finds the data in its local memory, appends this data to the header sent by the kernel, calculates the required checksums, and sends the response over the network. Such a system reduces traffic on the local interconnect by transferring only relatively short header information when possible, rather than transferring entire files for every HTTP request. While current zero-copy I/O systems avoid copying at the host CPU and main memory, this scheme further extends zero-copy I/O to the server's network interface.

Due to space, power, and cost constraints, the memory on a NIC is far more limited than the server's memory, so the network interface data cache will be much smaller than the operating system's file cache. Therefore, for effective caching, web server requests must have significant data locality. Figure 3 shows the percentage of the HTTP content traffic that would be eliminated from the local interconnect by network interface data caches of varying sizes. This figure was generated using a cache simulator that simply plays back a web trace and determines what portion, if any, of each successive requested file is currently in the cache. The web traces are from Berkeley's CS department, IBM, NASA, Rice's CS department, and the web site for the 1998 Soccer World Cup. The figure shows results for caches sized from 64 KB to 16 MB, which could easily be supported by DRAM on a NIC, with 4 KB blocks using least-recently used (LRU) replacement. The solid lines show the potential traffic reduction for a single cache. The dashed lines show the potential traffic reduction if two caches of the same size are used with the trace split evenly across the two, simulating a system with independent caches on two network interfaces. Even though this situation doubles the total cache size, the traffic reduction is slightly lower since splitting the traces reduces temporal locality. The figure shows that even modestly sized data caches can significantly reduce traffic from HTTP content, indicating substantial potential main memory and local interconnect bandwidth savings.

As shown in the figure, the World Cup and IBM traces have small working sets, so even 2 MB caches reduce HTTP content traffic by over 35%. The NASA and Rice traces begin to show significant benefit with 8 MB caches, with content traffic reductions of 40–45%. The potential traffic reduction for these four traces continues to improve as cache size increases, with 52–84% of HTTP content traffic elimi-

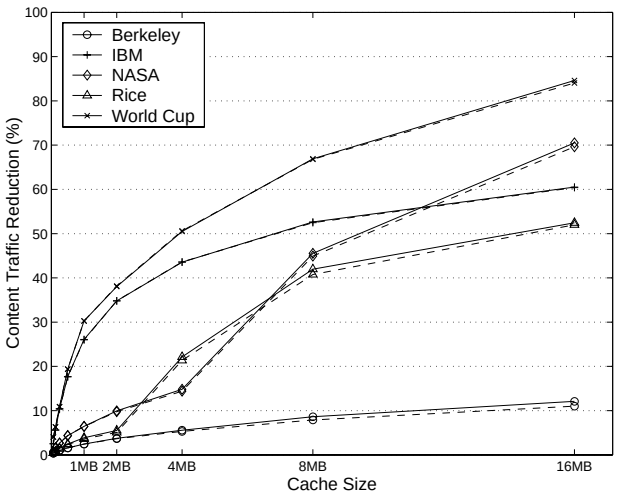

Figure 3: Potential reduction in HTTP content traffic using LRU caches. The X-axis depicts cache sizes from 64 KB to 16 MB, and the Y-axis shows the percentage of response content bytes eliminated from the local interconnect for each of the five traces. The solid and dashed lines show the content traffic reduction possible with one cache and two independent caches, respectively.

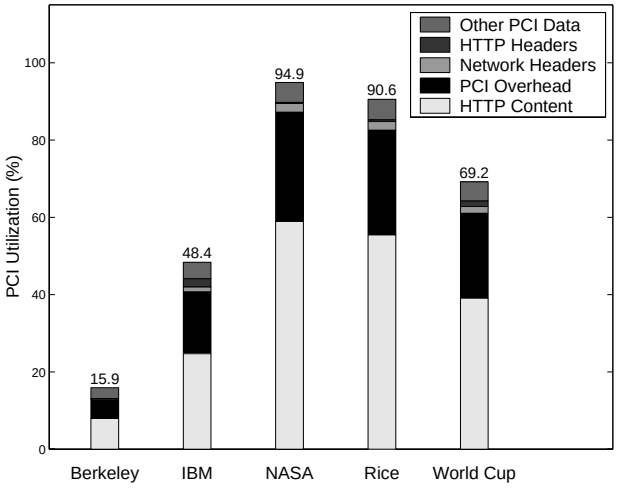

Figure 4: Measured PCI bus utilization for each trace, split into categories for HTTP content, PCI overhead caused by addressing and stalls, networking headers related to TCP/IP and Ethernet, HTTP-level headers, and other PCI data (e.g., traffic from other peripherals). Network interface data caching addresses the two largest components, HTTP content and PCI overhead.

nated from the local interconnect given dual 16 MB caches. The Berkeley trace, however, has a large working set, and therefore caches as large as 16 MB only reduce HTTP content traffic by 12%. Further evaluation shows that least-frequently used (LFU) and first-in first-out (FIFO) block cache replacement policies perform similarly to LRU, with LFU slightly better and FIFO slightly worse.

Figure 4 shows the utilization of the PCI bus during the execution of each workload on an actual PC-based web server. The server includes two Gigabit Ethernet NICs, a 64-bit/33 MHz PCI bus, and a VMETRO PBT-615B PCI bus analyzer for passively measuring PCI bus traffic. (Section 5 gives additional details about the experimental methodology.) The utilization shown represents the ratio of actual PCI traffic during the execution of the workload to the peak theoretical traffic for the server's PCI bus during the same time period. The figure categorizes the different sources of PCI utilization on the server, including HTTP response content, PCI overhead, networking headers (TCP/IP and Ethernet headers from the server), HTTP response headers, and other PCI data (including HTTP request headers, TCP/IP packets from the client, and traffic from other peripherals). PCI overhead accounts for bus cycles spent on transfer-related overheads such as address cycles, wait cycles caused by initiators or targets that are not ready to transfer data, or wait cycles on reads switching the bus from the address phase to the data phase. The server sees over 90% PCI bus utilization for the NASA and Rice traces, saturating the interconnect. The average requested file sizes for the IBM and World Cup traces are quite small compared to the other traces, causing the server CPU to saturate before the PCI bus. The CPU can only support enough throughput to utilize 48% and 69% of the PCI bus on the IBM and World Cup traces, respectively. The Berkeley trace requires heavy disk accesses due to its large working set size, so disk latency becomes a bottleneck. The HTTP content and PCI overhead account for around 60% and 30% of all PCI traffic, respectively. Thus, even at 95% utilization, a 64-bit/33 Mhz PCI bus only transfers 1.2 Gb/s of HTTP content on these traces.

Network interface data caching directly targets the largest component in all traces, HTTP content. Reductions in HTTP content traffic lead to reductions in the second largest component, PCI overhead, since the system will now see far fewer transfers. Caching should thus provide substantial overall PCI traffic reductions for these workloads since it addresses the components that account for roughly 90% of PCI traffic and achieves large reductions in content traffic with reasonable storage capacity.

4. Network Interface Data Cache Design

A network interface data cache utilizes a network interface with a programmable processor and a modest amount of on-board DRAM. The cache stores data that may be appended to packet-level and application-level headers generated by the host processor and then sent out over the network. The operating system on the host CPU determines which data to store in the network interface cache and for which packets it should use data from the cache.

4.1 Cache Architecture

The network interface data cache is simply a region of local memory on the network interface card. Upon initialization, a programmable network interface must allocate storage for

transmit and receive buffers, firmware code, and temporary storage needed by computations used to transmit and receive packets on the network. Any additional space can be used for the network interface data cache. The NIC processor notifies the operating system of the size of the network interface data cache after it has been allocated. Ideally, the network interface data cache is as large as possible, but, as shown in Figure 3, even modestly sized caches of a few megabytes can significantly reduce interconnect traffic.

Since the network interface data cache resides in the local memory of the network interface card, it may only be accessed by the NIC processor. However, the network interface data cache is controlled entirely by the operating system on the host processor. The NIC processor acts as a slave, inserting and retrieving information from the cache only at the direction of the host.

When adding new data to the network interface data cache, the operating system sends that data to the NIC and tells it to store the data at a particular offset in the cache. When the operating system decides to use data within the network interface data cache for a packet, it simply instructs the NIC processor to append data from the cache to the packet by giving the offset and length of the desired data. In this way, the operating system can use any data in the cache for any outgoing packet. For example, the data can be a subset of a block that was previously inserted into the cache or can straddle multiple cached blocks.

4.2 Cache Management

Since the NIC processor does not interpret the data in any way, the host processor must establish policies for allocation, replacement, and use of data in the network interface data cache. Additionally, the host processor must resolve the cache coherence problem that arises on modifications to the main memory copy of content replicated on the NIC. The operating system implements all policies for these cache management tasks.

When allocating storage in the network interface data cache, the operating system caches content at the granularity of a file block. Caching blocks instead of packets allows the TCP/IP stack to structure packet contents differently for different responses, if necessary, and also simplifies cache management by using fixed size objects. The operating system also manages a directory of the contents stored within the network interface data cache. The directory entries contain information relating a block in the NIC cache to the original file identifier, the offset within the file, the file revision identifier (maintained by the operating system to track changes in files), and any required status information associated with the data stored in the network interface data cache. A system with multiple NICs has separate directories for each network interface data cache, since the NICs have separate storage.

The operating system attempts to use data from the network interface data cache in response to the `sendfile` system call from the application. `Sendfile` is a commonly-implemented API for zero-copy I/O in servers. Although a server can also transfer data using the `read` and `write` system calls, the use of user-level data buffers in those system calls commonly causes the kernel to copy data from the kernel file cache to user space on a `read` and from user space to the kernel network buffers on a `write`. Such copying increases CPU and memory load in performing the copies and memory pressure in storing multiple copies of the same information. In contrast, `sendfile` allows for a straightforward implementation of zero-copy I/O since it refers to file content through a descriptor rather than a user-level buffer.

Figure 5 depicts the actions taken by the operating system in response to `sendfile`. If a call to `sendfile` specifies a portion of the file that resides in the operating system file cache, then the operating system creates a set of small memory buffers (called `mbuf`s in FreeBSD) to hold control information and a pointer to the data in the file cache. The operating system annotates these `mbuf` structures with the original file identifier, the offset into the file, and the file revision number (step 1). The `mbuf` chains created by `sendfile` are transformed into packets by the networking stack. Each `mbuf` can reference at most one contiguous region of memory, and the process of forming packets may split the `mbuf`s so that they reference subranges of pages (step 2). The networking stack then passes the `mbuf` chain for each packet to the device driver for the NIC that will be used for the transmission.

The device driver looks up each referenced block in the network interface data cache directory (step 3 of Figure 5). If the block is already present, then the driver simply transmits the offset and length of the content within the network interface data cache. If the block is not present, the driver seeks to allocate a block in the cache, using an LRU replacement policy to evict old blocks if no space is available. In either case, the driver creates a set of *buffer descriptors* to pass the relevant information to the NIC (step 4). Each buffer descriptor either points to a buffer in main memory or a region of the network interface data cache. The CPU typically notifies the NIC that it has created new buffer descriptors by writing to a memory mapped register of the NIC. The NIC then retrieves the buffer descriptors using DMA and uses the information contained within them to initiate the necessary DMA transfers to retrieve the data from main memory. After completing the requested operation, the NIC interrupts the CPU to inform it that the buffer descriptors have been consumed. In a system without a network interface data cache, these buffer descriptors always require the NIC to transfer packet data from main memory using DMA. In a system with a network interface data cache, however, buffers pointing to cached content do not require a DMA transfer for packet data.

The file revision field stored in each directory entry enables a straightforward mechanism to keep the cached blocks coherent with the objects stored on the server's main storage sys-

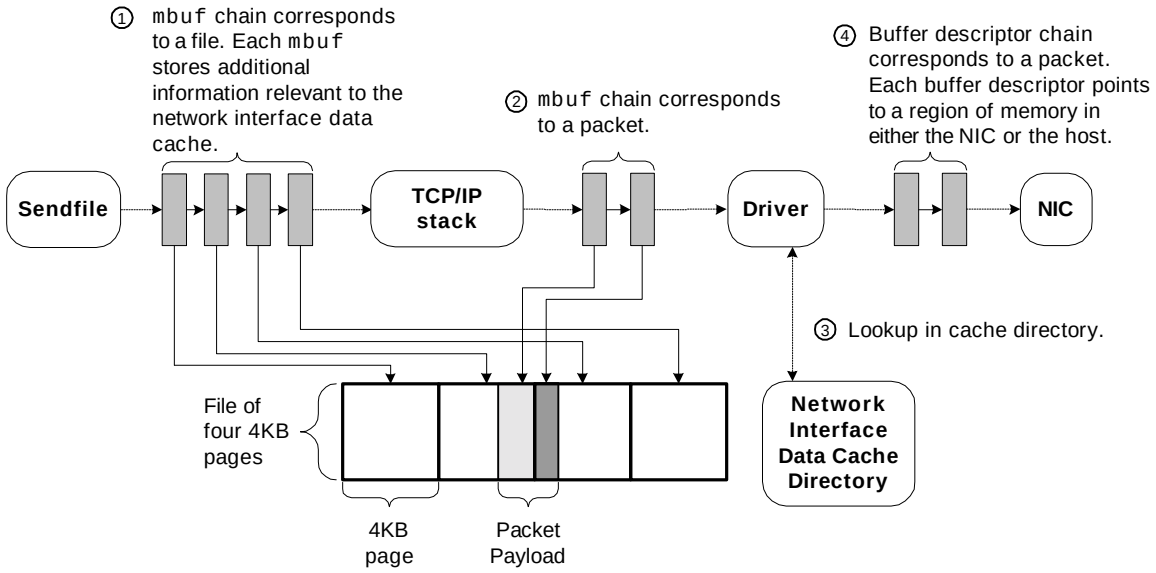

Figure 5: Steps in sending a response using the `sendfile` system call. In step 1, `sendfile` creates a chain of `mbuf`s pointing to the pages in the file. In step 2, the TCP/IP stack splits the file into packets, creating a new `mbuf` chain for each packet. In this step, each `mbuf` points to a subrange of a page. There is also an `mbuf` at the head of the chain (not shown) for the TCP/IP headers. In step 3, the NIC's driver consults the network interface data cache directory to determine if any parts of the packet are cached. Finally, in step 4, the driver creates a chain of buffer descriptors for each packet. Each buffer descriptor points to data in either main memory or the network interface data cache.

tem. When looking up blocks in the network interface data cache directory, the device driver lazily invalidates blocks for which the current revision identifier does not match the cached revision. Note that multiple NICs do not present any additional cache coherence problems, since each network interface data cache operates independently.

To further simplify the coherence implementation, the operating system keeps the main memory block cache strictly inclusive of the network interface data cache. With this guarantee of inclusion, information mapping file blocks to network interface data cache storage need not persist beyond the replacement of a block from the main memory file cache. This inclusion property provides two further benefits. First, it allows caching even for NICs that do not support checksum offloading, since all content on the NIC also resides in the main memory and can thus have checksum calculations performed by the host CPU. Second, inclusion simplifies network retransmits in the event of replacements from the network interface data cache, since the block cache already includes mechanisms to support retransmits for applications that use the `sendfile` API.

4.3 Cache Interface

The operating system manages the network interface data cache using an API that consists of the four functions listed in Figure 6. The table only shows the functions used to send packets; the NIC must also support functions to receive packets and perform other actions, but those need not be modified since the cache does not address those tasks. Since the host processor cannot directly call functions on the network processor, these API functions are actually implemented using existing mechanisms to communicate from the host processor to the NIC. In particular, the operating system uses flags in the buffer descriptor communication structures described in Section 4.2 to indicate which command to invoke, and additional fields within those buffer descriptors to pass arguments to the NIC.

The API for the network interface data cache includes functions to initialize the cache, to copy data from main memory to the network interface data cache, to append a block of main memory to the current packet, and to append a cached block to the current packet. All other NIC functions remain unchanged. The initialization function, `nic_cache_init`, allocates space in the NIC's local memory and notifies the operating system of the amount of memory that has been allocated so that the operating system may construct and manage the cache directory. Data is added to the cache using the `nic_cache_insert` function. Data is transferred from the main memory to the network interface data cache using DMA. As with all DMA transfers, a single buffer descriptor can only describe a contiguous buffer. So, if disjoint memory regions are to be added to the cache, the operating system must call `nic_cache_insert` multiple times.

The API of a conventional NIC effectively only includes the `nic_pkt_append_mm` function to construct and send packets. As described in Section 4.2, packets are transmitted over the network by generating a list of buffer descriptors that are then sent to the NIC. Each buffer descriptor points to a region of main memory that the NIC should ap-

- `nic_cache_init()`
  Allocate and initialize the data cache on the programmable network interface and return its size.
- `nic_cache_insert(mmaddr, ncoffset, len)`
  The buffer descriptor contains `mmaddr`, `ncoffset`, and `len`. Use DMA to transfer `len` bytes starting at main memory address `mmaddr` into the network interface data cache starting at offset `ncoffset`.
- `nic_pkt_append_mm(mmaddr, len)`
  The buffer descriptor contains `mmaddr`, `len`, and additional flags. Use DMA to transfer the `len` bytes starting at main memory address `mmaddr` into the network interface's transmit buffer. This function is unchanged from the original operation of the NIC.
- `nic_pkt_append_cache(ncoffset, len)`
  The buffer descriptor contains `ncoffset`, `len`, and additional flags. Copy `len` bytes from the cache starting at offset `ncoffset` into the transmit buffer.

Figure 6: Commands supported by the programmable processor on the network interface and invoked by the operating system. These commands are passed to the network interface through the buffer descriptors, which are currently used to control DMA transfers.

pend to the current packet by using DMA to transfer that block of memory from the host to the transmit buffer. This is accomplished by having each buffer descriptor invoke the `nic_pkt_append_mm` function on the NIC. Additional flags in the buffer descriptor are used to indicate to the NIC if that block is the first or last block in the packet, if a particular function should be performed before sending the packet (such as checksum offloading or other future services), or any additional information required by the NIC to process the packet.

The last API function is `nic_pkt_append_cache`. This function resembles `nic_pkt_append_mm` but copies data to the transmit buffer from the indicated offset in the network interface data cache instead of using a DMA from main memory. Note that even this internal copy could be eliminated if the engine that transfers the data out onto the network could gather the packet from disjoint memory regions on the NIC. As with `nic_pkt_append_mm`, additional flags in the buffer descriptors are used to indicate if the block is the last block in the packet or if additional processing should occur.

Referring back to Figure 5 in Section 4.2, the buffer descriptors of step 4 convey the network interface data cache commands of Figure 6. If the data represented by an `mbuf` is not present in the network interface data cache, the driver inserts the data into the cache using `nic_cache_insert`. The driver then sends a series of buffer descriptors of type `nic_pkt_append_cache` or `nic_pkt_append_mm`, as appropriate, to transmit each packet. Headers for TCP/IP, Ethernet, and HTTP are transferred to the NIC as before and are never inserted in the network interface data cache. The NIC processor concatenates the cached data with the headers fetched from the host memory before the packets are transmitted onto the network.

Note that packets can bypass the network interface data cache by only using `nic_pkt_append_mm`. Such packets are useful in circumstances where the driver chooses not to cache a file block or where a transmission does not refer to a file block (such as dynamic content, `ping`, or `telnet`).

5. Experimental Methodology

5.1 Prototype Implementation

This section describes a prototype implementation of network interface data caching in a PC-based server. The prototype implements an LRU block cache with lazy invalidation, a block size equal to the page size of the operating system (4 KB), and a requirement of inclusion in the main memory file cache. The server has an AMD Athlon XP 2200+ processor, a 64-bit/33 Mhz PCI bus, 2 GB of DDR SDRAM, two 36 GB SCSI disks, two 3Com 710024 copper Gigabit Ethernet NICs, and a VMETRO PBT-615B PCI bus analyzer. The 3Com 710024 NIC is based on the Tigon 2 chipset, with two 88 Mhz MIPS R4000-based processors. Each NIC also has 1 MB of on-board memory. The PCI bus analyzer passively measures the actual PCI bandwidth utilization and incurs no PCI overhead.

The server runs a slightly modified FreeBSD 4.6 operating system, with the `sendfile` system call extended to use network interface data caching. Support for network interface data caching requires the addition of five new fields to the `mbuf` structure, about 150 modified lines in the `sendfile` system call and `mbuf` manipulation routines, and roughly 850 lines of new code in the device driver for the network interface.

The network interface in the server uses a modified version of Revision 12.4.13 of the Tigon firmware, which was made open-source by the manufacturer [1]. The modified firmware implements the API commands of Section 4.3 and various optimizations to eliminate the possibility of the NIC being a potential bottleneck. Specifically, the modified firmware parallelizes tasks across the two processors and sets tunable parameters to communicate with the host more efficiently.

The 3Com NIC only has 1 MB of on-board memory, which is insufficient for our experiments, so the prototype firmware emulates various cache sizes. Upon receiving a `nic_cache_insert` command, the prototype fetches the specified data and discards it instead of adding it to the cache. On a `nic_pkt_append_cache` command, the prototype simply increments the pointer to the end of the transmit buffer by the specified length, using whatever data is currently in the buffer. The other API functions behave as specified. The NIC with these simplifications generates the same amount of PCI bus traffic and Ethernet traffic as a fully

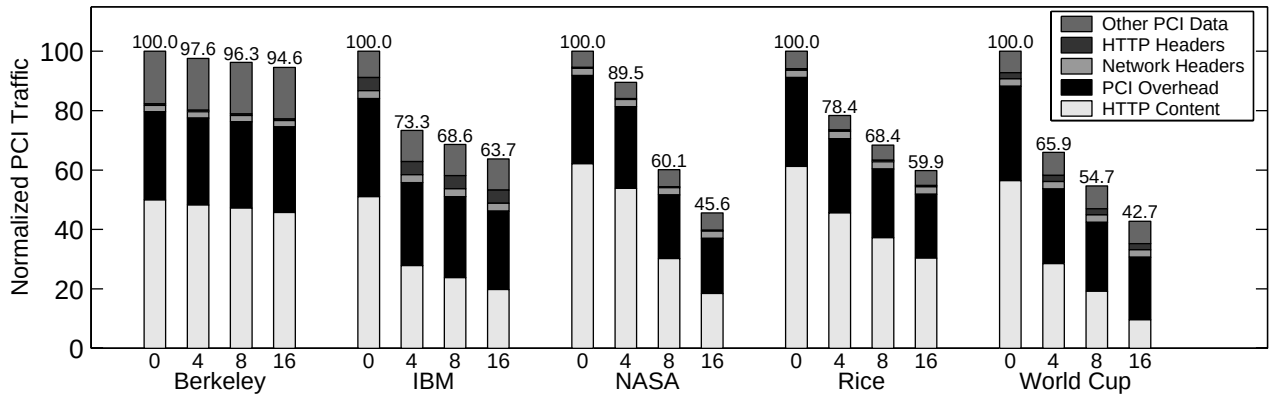

Figure 7: Comparison of PCI Traffic with and without network interface data caching. The cache sizes are 4, 8, and 16 MB per network interface. The cache size 0 MB means that caching is not used.

functional NIC that actually stores cached blocks and reuses them on appropriate commands. However, the lack of copying in `nic_pkt_append_cache` understates the overhead in some cases, but is reasonable for a NIC that supports gather I/O. Furthermore, packets that include cached data have invalid checksums. The Tigon checksumming hardware is integrated into the DMA engine, so only data that is transferred between the host and the Tigon may be checksummed. A slight modification to the Tigon architecture to allow data stored in local memory to be run through the checksumming hardware, such as allowing a DMA transfer to read from the local memory and then discard the data, would solve this problem. These simplifications to the prototype allow the use of a current programmable NIC with insufficient memory to evaluate network interface data caching.

5.2 Test Platform

The performance testbed consists of the prototype web server and two client machines connected via a Gigabit Ethernet switch. Each client machine has an AMD Athlon MP 1800+ processor and two Intel Pro/1000 MT Gigabit Ethernet NICs. The client machines run FreeBSD 4.5. The server and clients are connected through a Netgear GS508T Gigabit Ethernet switch isolated from the rest of the network, so there is no background traffic during the experiments. The experiments use two private subnets, and each machine in the testbed has one network interface on each subnet.

The network interface drivers on the synthetic clients have been modified to accept packets from the prototype server containing cached data despite the invalid checksums discussed in Section 5.1; to support this distinction, the server marks such packets with an artificial time-to-live field in the IP header. Such modifications would not be needed if the network interface had sufficient memory to properly support `nic_pkt_append_cache`.

The application-level web server used is a modified version of thttpd 2.22beta4, an efficient event-driven web server [15]. The modifications include the use of the `sendfile` API, support for HTTP pipelined persistent connections, and other generally applicable optimizations.

Server throughput is measured using a trace replayer tool, which takes a web trace as input and simulates multiple users by opening multiple simultaneous TCP connections to the server, each of which corresponds to a single user. The trace replayer uses an infinite-demand model, issuing requests as fast as the server can sustain responses. Requests that came from the same anonymized client IP address within a fifteen second period in the original access log are treated as a single persistent connection. Within a persistent connection, requests that arrive less than five seconds apart in the original log are pipelined. Each client machine runs two instances of the trace replayer, which connect to the server through different subnets. The traces are split equally among all four replayers. As described in Section 3, the web traces are from Berkeley's CS department, IBM, NASA, Rice's CS department, and the web site for the 1998 Soccer World Cup. The traces include between 300 thousand (Rice) and 15 million (IBM) requests.

6. Experimental Results and Discussion

As shown in Figure 3 of Section 3, there is significant locality of access to web pages in a typical server. Data caches of 16 MB per network interface have the potential to eliminate 52–84% of HTTP content traffic from the local interconnect for four of the web traces and 12% for the Berkeley trace. This section discusses the interconnect traffic reductions achieved in the prototype system and their effect on server throughput, as well as other possible designs and deployments.

6.1 Local Interconnect Traffic

Figure 7 shows the impact of network interface data caching on PCI traffic, with four bars for each trace. The leftmost bar represents traffic without caching, and the right three bars represent traffic with cache sizes of 4 MB, 8 MB, and 16 MB

Trace	Cache Size per NIC	Reduction in HTTP Content	Reduction in PCI Overhead	Throughput (Mb/s)	Request Rate (req/s)	Cache Benefit
Berkeley	none			161	828	
	4 MB	4%	2%	160	822	-1%
	8 MB	6%	2%	160	821	-1%
	16 MB	9%	3%	160	820	-1%
IBM	none			502	22777	
	4 MB	44%	15%	500	22691	0%
	8 MB	52%	17%	501	22743	0%
	16 MB	60%	20%	501	22726	0%
NASA	none			1198	2755	
	4 MB	13%	8%	1294	2978	8%
	8 MB	51%	28%	1509	3472	26%
	16 MB	70%	38%	1571	3614	31%
Rice	none			1126	4361	
	4 MB	27%	17%	1253	4869	11%
	8 MB	40%	23%	1287	5021	14%
	16 MB	51%	28%	1322	5152	17%
World Cup	none			794	15190	
	4 MB	49%	21%	813	15545	2%
	8 MB	66%	27%	830	15889	5%
	16 MB	83%	34%	849	16246	7%

Table 1: Improvement in server performance from caching. The measured throughput includes only actual HTTP content, not HTTP or networking headers.

per network interface. All measures of PCI traffic are normalized to the traffic without caching. As in Figure 4, the bars are each split into five categories. As can be seen in Figure 7, network interface data caching reduces the amount of HTTP content transferred across the local interconnect substantially for all workloads except Berkeley, as predicted by Figure 3. Reducing the bus traffic from HTTP content also reduces the associated PCI bus overhead by eliminating some addressing cycles and transfer stalls.

The third and fourth columns of Table 1 show the actual reductions in PCI bus utilization from HTTP content traffic and PCI overhead for each trace and cache size listed in the first two columns. As the content traffic reduction increases, the PCI overhead decreases, leading to further reductions in PCI traffic. As anticipated in Section 3, this trend indicates that reducing the actual data transfers also effectively reduces the PCI overhead. This overhead was originally the second largest part of PCI utilization, at roughly 30%.

With 16 MB data caches on each network interface, the server reduces HTTP content traffic on the bus by 51–83% for the IBM, NASA, Rice, and World Cup traces. These numbers closely match the predictions in Figure 3 despite being measured on a real system, which may reorder requests and responses due to various latencies in the system and the different ways of splitting the traces. The PCI overhead accordingly decreases by 20–38%, combining for overall PCI bus traffic reductions of 36–57%. Since only HTTP content is cached on the network interface, network interface data caching does not change the other types of PCI traffic such as HTTP and network headers.

The Berkeley workload shows minimal reduction in overall PCI traffic because its large working set size allows network interface data caching to eliminate only 9% of the HTTP content traffic from the bus. More intelligent replacement policies may provide additional benefits for such a workload [4, 6]. Additionally, predicting reuse patterns could allow for reducing cache pollution by bypassing the cache entirely for some data, as has been studied in other contexts [16].

These reductions in PCI traffic have several consequences. First, systems that are limited by the achievable PCI bandwidth will see an improvement in web server performance commensurate with this reduction in traffic. Second, systems that are not limited by the achievable PCI bandwidth will be able to scale other resources in the system beyond the limits currently imposed by the local interconnect. In both cases, the potential to extract greater performance from existing shared interconnects makes more radical changes to local interconnect designs less attractive because of the additional engineering costs they impose in redesigning motherboards, peripheral interfaces, interconnection components, and operating systems.

6.2 Server Throughput

The final three columns of Table 1 show the throughput improvements achieved by network interface data caching for the various traces and cache sizes. As shown in Figure 4, the PCI bus is saturated for the NASA and Rice traces without caching. Therefore, they benefit the most from network interface data caching. Table 1 shows that the server achieves a 31% throughput improvement on the NASA trace and a 17%

improvement on the Rice trace using 16 MB caches in the network interfaces. This enables the server to achieve a peak throughput of 1571 Mb/s on the NASA trace and 1322 Mb/s on the Rice trace. Network interface data caching is most effective at capturing the locality of the World Cup trace, with 16 MB caches reducing 83% of the bus traffic for HTTP content. However, since PCI utilization is only 69%, this translates to a smaller throughput improvement than on the NASA and Rice traces. As CPU speeds increase, servers will be able to completely saturate the PCI bus on this trace, and then the high PCI traffic reduction from network interface data caching should translate into more significant throughput improvements.

Unsurprisingly, the server sees no throughput improvement from caching for the Berkeley and IBM traces. As discussed in Section 3, disk latency limits server throughput for Berkeley, and CPU performance limits throughput for the IBM trace. PCI utilization is thus low for both traces, so any reduction in PCI traffic does not improve throughput. Additionally, the overhead of managing the cache and using the cache commands causes a slight performance degradation.

6.3 Operating System Design Alternatives

A more radical design of the operating system could better exploit knowledge of the network interface data cache to shape performance decisions differently. For example, systems with multiple NICs are often *multi-homed*, with addresses on separate networks and multiple routes available to them to reach certain destinations. Such systems can use routing protocol information to construct a table of routing metrics to different networks. In contrast, this study uses multiple NICs on separate private subnets, leaving the server with no choice as to which NIC to use when sending data. The design of Section 4 assumes that the network interface is selected in a higher level of the networking stack, with the NIC driver then responsible for determining if the data is cached. A more integrated design could use information about the likelihood of holding data in one network interface data cache rather than another to bias the routing decision for a flow of packets. If this decision is made accurately, such a strategy may improve effective cache capacity by reducing the likelihood of replicating the same data in multiple network interface data caches.

The prototype maintains cache coherence by lazily invalidating cached blocks that do not match the current file revision identifier. Other alternatives for coherence include update-based protocols that copy modified blocks to the cache or eager invalidation protocols that remove a block from the cache directory as soon as a modification takes place. Update protocols may require extra PCI traffic if multiple writes take place before the data is read again. On the other hand, eager invalidation may allow for more intelligent cache replacement policies if modified blocks are freed more rapidly. Both variants require additional code complexity since the operating system must also observe actions that write to cached blocks. However, such variants may be required in applications or deployments without valid revision identifiers.

6.4 Deployment in Other Applications

Network interface data caching has potential value beyond improving the throughput of web servers. Any networking server that sends repeated responses to frequent requests could make use of network interface data caching. While this paper focuses on web servers using TCP/IP, other application servers and protocols could also benefit from the technique. To be useful, the system must exhibit locality in the responses to network requests and have high enough bandwidth demands that local interconnect traffic is a bottleneck. For example, NFS and streaming media servers could potentially benefit from caching data within the network interface and reducing traffic over the local interconnect.

An NFS server potentially sends the same file out over the network several times without the file being modified, allowing caching at the network interface. However, NFS servers introduce additional complexity into the coherence protocol, as files may be updated remotely. The design must then make choices about whether to invalidate cached data on a write, have the host update the cached copy, or have the network interface recognize that incoming packets are updating data within its cache.

Streaming media servers come in two basic flavors: live broadcasts and on-demand streaming. Although a live stream has no temporal reuse of data, multiple simultaneous clients receive the same content at the same time. In this model, the host processor would send a live block to the network interface data cache once. The clients viewing the stream would then have this block served to them from the cache, with IP and either TCP or UDP headers generated by the host. The block could be replaced immediately after delivery, since a live stream has no temporal locality beyond the length of a single frame of data. The potential benefit of caching is substantial since such a server is likely to be bandwidth bound, increasing the likelihood of a PCI bus bottleneck. Media servers for previously-recorded on-demand content may see benefits akin to those seen in web servers from repeated access to the same streams, particularly for short files such as MP3 audio files and video advertisements. However, full-length videos may have impractically large file sizes for any substantial caching.

7. Related Work

Various studies have addressed the bottleneck of CPU and main memory through zero-copy I/O, both with and without modifications to the underlying system call API [7, 11, 13]. Although the current prototype of network interface data caching uses the `sendfile` API, other implementations of zero-copy I/O should provide sufficient information to support network interface data caching. In particular, network interface data caching can obtain the directory infor-

mation it requires from any implementation that shares data structures between the file cache and the network buffers.

Yocum and Chase recently proposed payload caching for network intermediaries (e.g., firewalls and routers) [18]. A payload cache stores incoming packet payload data directly in the network interface to avoid sending such content back to the network interface for retransmission, while imposing few constraints on the NIC processor and using the host CPU to generate headers. Payload caching uses storage on the NIC as a short-term holding buffer and associates that storage with information specific to a packet in transit. Network interface data caching adapts this concept to the server domain by caching content that originates locally rather than caching incoming network traffic. Network interface data caching also manages the storage more flexibly to facilitate reuse across packets and connections that may choose different packet structures in different situations.

Recent work has also considered the use of programmable network interfaces to improve the performance of application-level networking in other contexts. For example, Buonadonna and Culler provide a low-latency non-socket interface for communication in system area networks by offloading a subset of the network stack to a Myrinet LANai 9 network interface [5]. Krishnamurthy et al. use I2O network interfaces with i960 processors to control disks and stream data directly from disk to the network without using the host CPU, PCI bus, or main memory [10]. Petrini et al. use the Elan network interface to provide a single address space and implement the MPI library for communication in a distributed computing environment, using the DRAM on the network interface as an extended buffer for remote communication [14].

Previous studies have found high levels of locality in Web server traces [2]. This work confirms those findings. Furthermore, a variety of advanced replacement policies to exploit this locality have been proposed for the web file cache environment [4, 6]. Other work has considered replacement policies for network file server block caches [17]. The specific policy choices for allocation and replacement are independent of the concepts expressed here, so any policies could be adopted.

8. Conclusions

Repeatedly transferring frequently-requested data across an expensive local interconnect leads to an inefficient use of system resources. Furthermore, interconnects scale more slowly than processing power or network bandwidth because of the need for standardized interfaces across devices. Caching data directly on a programmable network interface reduces local interconnect traffic on networking servers by eliminating repeated transfers of frequently-requested content. A prototype implementation of network interface data caching reduces PCI bus traffic by 36–57% on four web workloads with only 16 MB caches on two network interfaces. This technique allows application-level performance to scale with more aggressive CPUs and network links beyond the point at which less efficiently utilized local interconnects would become a bottleneck. Such reductions in interconnect traffic only require a modest amount of DRAM in the network interface and impose no constraints on the network interface's processor.

Network interface data caching only requires the addition of five fields to the `mbuf` structures that refer to kernel data buffers, about 150 modified lines in the `sendfile` system call and `mbuf` manipulation routines, and roughly 850 lines of new code in the device drivers for the network interface. These simple additions to the operating system and 16 MB caches on two network interfaces enable web server throughput improvements of 7–31% for three web workloads studied, directly resulting from the reduction of data transfers from main memory to the network interface. Therefore, the introduction of a programmable network interface with 8–16 MB of DRAM would allow existing web servers to realize this throughput improvement immediately by more efficiently utilizing their local interconnects. Although the prototype uses the FreeBSD `sendfile` system call and a PC-based web server with a PCI bus, the concepts of network interface data caching are independent of the specific zero-copy I/O mechanism and the specific local interconnect.

Network interface data caching applies in other environments as well, since a variety of systems repeatedly transfer data across the network. Examples include NFS servers, streaming media servers, computation clusters, and network attached storage. While the access locality in each environment will be different, caching data within the network interface is a conceptually simple and practically implementable mechanism for exploiting that locality.

Acknowledgments

The authors thank Alan Cox and Mike Filippo for their valuable suggestions and comments about the work in general. Additionally, the authors thank Ujval Kapasi and Erich Nahum for comments on the paper. Erich Nahum also provided the IBM trace.

References

[1] Alteon WebSystems. *Gigabit Ethernet/PCI Network Interface Card: Host/NIC Software Interface Definition*, July 1999. Revision 12.4.13.

[2] Martin F. Arlitt and Carey L. Williamson. Internet Web Servers: Workload Characterization and Performance Implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, October 1997.

[3] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 253–265, June 1999.

[4] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Schenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE INFOCOM '99*, volume 1, pages 126–134, March 1999.

[5] Philip Buonadonna and David Culler. Queue Pair IP: A Hybrid Architecture for System Area Networks. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 247–256, May 2002.

[6] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, pages 193–206, December 1997.

[7] Peter Druschel and Larry L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP-14)*, pages 189–202, December 1993.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol - HTTP 1.1. IETF RFC 2616, June 1999.

[9] Karl Kleinpaste, Peter Steenkiste, and Brian Zill. Software Support for Outboard Buffering and Checksumming. In *Proceedings of the ACM SIGCOMM '95 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 87–98, August 1995.

[10] Raj Krishnamurthy, Karsten Schwan, Richard West, and Marcel-Cătălin Rosu. A Network Co-processor-Based Approach to Scalable Media Streaming in Servers. In *Proceedings of the 2000 International Conference on Parallel Processing*, pages 125–134, August 2000.

[11] Erich M. Nahum, Tsipora Barzilai, and Dilip Kandlur. Performance Issues in WWW Servers. *IEEE/ACM Transactions on Networking*, 10(2):2–11, February 2002.

[12] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 199–212, June 1999.

[13] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. I/O-Lite: A Unified I/O Buffering and Caching System. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, pages 15–28, February 1999.

[14] Fabrizio Petrini, Wu-Chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE MICRO*, 22(1):46–57, January 2002.

[15] Jef Poskanzer. *thttpd - tiny/turbo/throttling HTTP server*. Acme Labs, February 2000. Unix manual page.

[16] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 93–103, December 1995.

[17] Darryl L. Willick, Derek L. Eager, and Richard B. Bunt. Disk Cache Replacement Policies for Network Fileservers. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 2–11, May 1993.

[18] Ken Yocum and Jeff Chase. Payload Caching: High-Speed Data Forwarding for Network Intermediaries. In *Proceedings of the 2001 Annual USENIX Technical Conference*, pages 305–317, June 2001.